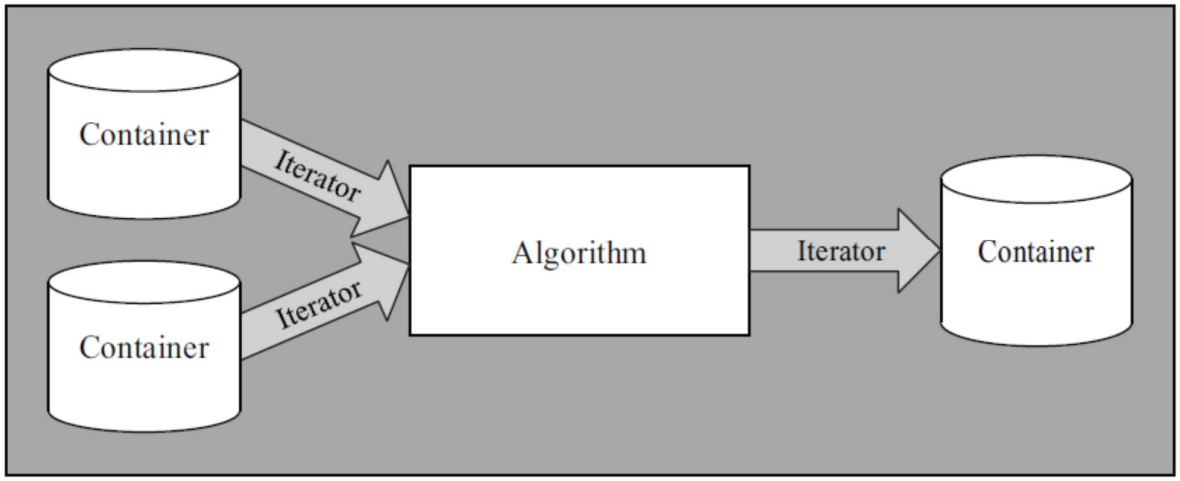
# C++ Standard Template Library (STL)

- Modern data processing often involves collections of objects being processed.
- C++ STL is a powerful framework for working with collections.
- Relief you from:
  - Details of data structures
  - Necessary memory management when processing collections.

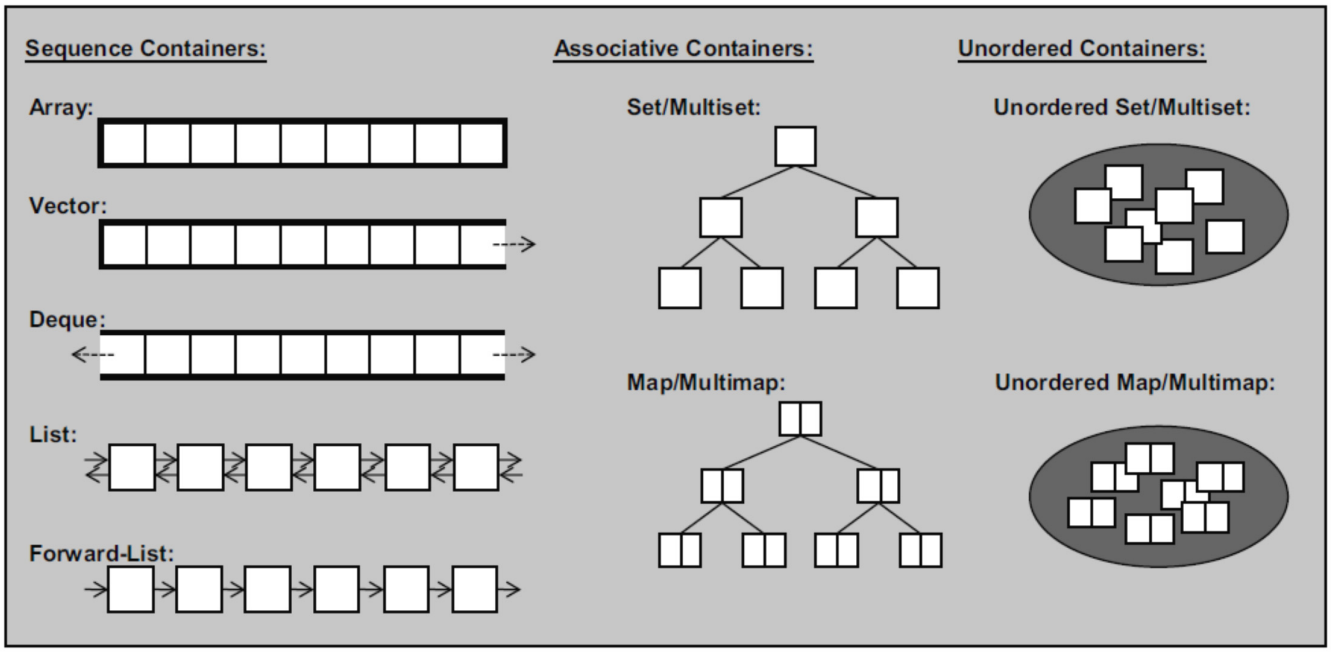
## STL Components: Containers, Iterators and Algorithms

- Containers (vector…): to manage collections of objects.
- Iterators: to step through the elements of containers.
- Algorithms: to process the elements of containers.



- **Sequential container**
  - The elements form a sequence in the order defined by the application programmer (you).
- **Associative container**
  - Elements are automatically sorted.
  - Value of the element determines its position.
- **Unordered container**

See Note



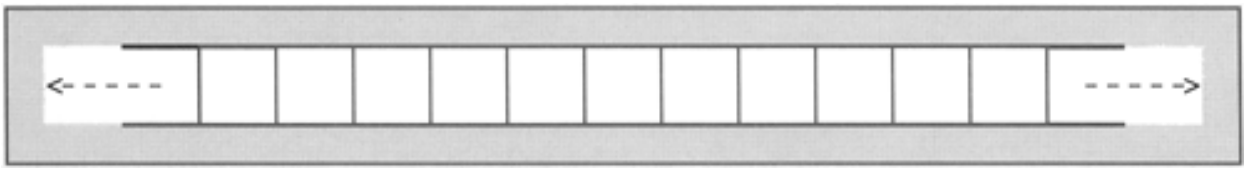

# Vector

- A container that manages its elements as a dynamic array.
- Elements can accessed **super-fast** using index operator [ ] (random access).
- Insert and erase elements are **super-fast** if these elements are located at the end of the vector container.

Simple usage, see note

# Deque (pronouced ‘deck’)

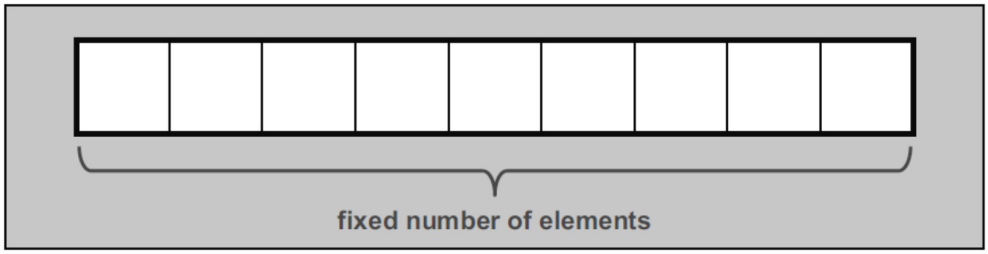
- Abbreviation of “double-ended queue”
- Dynamic array that can grow in both directions (push_back and push_front).



Simple usage, see note

# Array

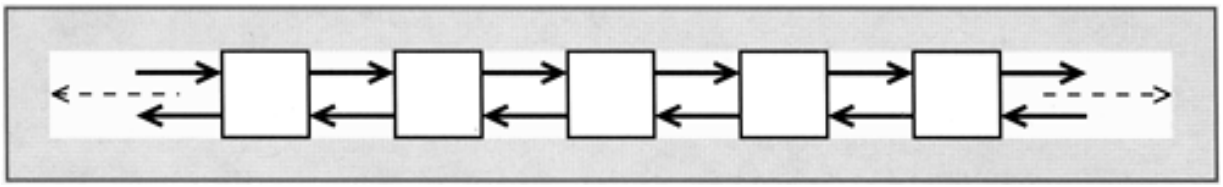
- Fixed-size array. Cannot add or remove elements dynamically.
- Wrap built-in array with STL interface without performance penalty.
- Elements can **super-super-fast** accessed using index operator [] (random access).




Simple usage, see note

# List

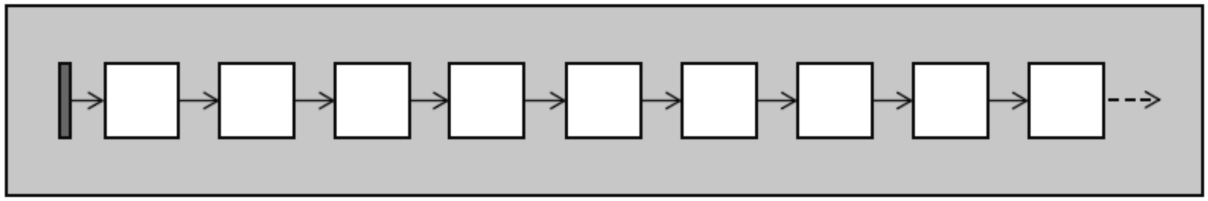
- **Double** linked list of elements.
- Every element in the collection refers to one predecessor and one successor.
- Insert and erase elements in the mid of list is fast (not super fast).



Simple usage, see note

# Forward List

- Singly linked list of elements.
- Every element in the collection refers to its successor only.
- Insert and erase elements in the mid of list is fast (not super fast).



see note

# Iterators

- STL containers are generally accessed using iterators.
- Iterators are pointer-like objects that can "iterate" containers.
- Every iterator represents a position in a container.

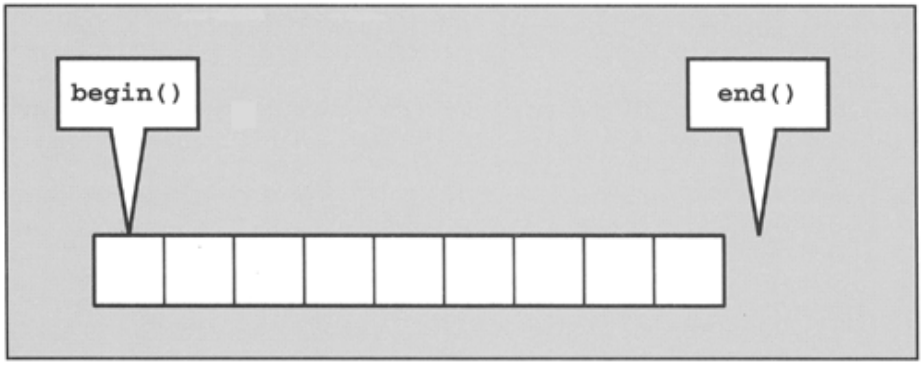
# Common Iterator Operations

| | |
|---|---|
| `*iter` | Return a reference to the element referred to by the iterator `iter`. |
| `iter->mem` | Dereference `iter` and fetch the member named `mem` from the underlying element. Equivalent to `(*iter).mem`. |
| `++iter iter++` | Increment `iter` to refer to the next element in the container. |
| `--iter iter--` | Decrement `iter` to refer to the previous element in the container. |
| `iter1 == iter2`<br>`iter1 != iter2` | Compare two iterators for equality (inequality). Two iterators are equal if they refer to the same element of the same container or if they are the off-the-end iterator (Section 3.4, p. 97) for the same container. |

# Additional Iterator Arithmetic (for string, array, vector, and deque only; **NOT** for list and forward_list)

| | |
|---|---|
| `iter + n`<br>`iter - n` | Adding (subtracting) an integral value `n` to (from) an iterator yields an iterator that many elements forward (backward) within the container. The resulting iterator must denote elements in, or one past the end of, the same container. |
| `iter1 += n`<br>`iter1 -= n` | Compound-assignment for iterator addition and subtraction. Assigns to `iter1` the value of adding `n` to, or subtracting `n` from, `iter1`. |
| `iter1 - iter2` | Subtracting two iterators yields the number that when added to the right-hand iterator yields the left-hand iterator. The iterators must denote elements in, or one past the end of, the same container. |
| `>, >=, <, <=` | Relational operators on iterators. One iterator is less than another if it refers to an element that appears in the container before the one referred to by the other iterator. The iterators must denote elements in, or one past the end of, the same container. |

# Iterator Range




- [begin, end): left-inclusive interval (standard C/C++ way, similar to zero-index rule in array)
- (Q1) when begin equals end, what does it imply on the range?
- (A1)
- When begin is not equal to end, there is at least one element in the range, and begin refers to the first element in that range.

Exercise (see note)

# Define and Initialize a Container

| | |
|---|---|
| `C c;` | Default constructor. If `C` is `array`, then the elements in `c` are default-initialized; otherwise `c` is empty. |
| `C c1(c2)`<br>`C c1 = c2` | `c1` is a copy of `c2`. `c1` and `c2` must have the same type (i.e., they must be the same container type and hold the same element type; for `array` must also have the same size). |
| `C c{a,b,c...}`<br>`C c = {a,b,c...}` | `c` is a copy of the elements in the initializer list. Type of elements in the list must be compatible with the element type of `C`. For `array`, the list must have same number or fewer elements than the size of the `array`, any missing elements are value-initialized (§ 3.3.1, p. 98). |
| `C c(b, e)` | `c` is a copy of the elements in the range denoted by iterators `b` and `e`. Type of the elements must be compatible with the element type of `C`. **(Not valid for `array`.)** |

**Constructors that take a size are valid for sequential containers (not including `array`) only**

| | |
|---|---|
| `C seq(n)` | `seq` has `n` value-initialized elements; this constructor is `explicit` (§ 7.5.4, p. 296). **(Not valid for `string`.)** |
| `C seq(n,t)` | `seq` has `n` elements with value `t`. |

see note

# Add Elements to a Sequential Container

**These operations change the size of the container; they are not supported by `array`.**
**`forward_list` has special versions of `insert` and `emplace`; see § 9.3.4 (p. 350).**
**`push_back` and `emplace_back` not valid for `forward_list`.**
**`push_front` and `emplace_front` not valid for `vector` or `string`.**

| | |
|---|---|
| `c.push_back(t)` | Creates an element with value `t` or constructed from *args* at the end of `c`. Returns `void`. |
| `c.emplace_back(args)` | |
| `c.push_front(t)` | Creates an element with value `t` or constructed from *args* on the front of `c`. Returns `void`. |
| `c.emplace_front(args)` | |
| `c.insert(p,t)` | Creates an element with value `t` or constructed from *args* before the element denoted by iterator `p`. Returns an iterator referring to the element that was added. |
| `c.emplace(p, args)` | |
| `c.insert(p,n,t)` | Inserts `n` elements with value `t` before the element denoted by iterator `p`. Returns an iterator to the first element inserted; if `n` is zero, returns `p`. |
| `c.insert(p,b,e)` | Inserts the elements from the range denoted by iterators `b` and `e` before the element denoted by iterator `p`. `b` and `e` may not refer to elements in `c`. Returns an iterator to the first element inserted; if the range is empty, returns `p`. |
| `c.insert(p,il)` | `il` is a braced list of element values. Inserts the given values before the element denoted by the iterator `p`. Returns an iterator to the first inserted element; if the list is empty returns `p`. |


Adding elements to a `vector`, `string`, or `deque` potentially invalidates all existing iterators, references, and pointers into the container.

see note

# Until Next Time …

- First midterm statistics.
- Lab4 starts at 6:00 pm on Thurs (10/23).
- HW4 will be issued on Thurs and due at 2100 next Wed.
- Read Chapters 9 and 10 carefully.